HW1_yigit_catak

November 7, 2020

1 Solving Bloxorz using Search

1.1 Introduction, Reading Board and Finding Important Coordinates in the Board

Defining used librariers and a function to read the game from the txt file

[ ]:
```
import queue
import time

def ReadBoard(text):
  targetBoard = "/content/drive/My Drive/CS 404/HW1/" + text
  File = open(targetBoard, 'r')

  Lines = File.readlines()

  xs1 = -1
  xs2 = -1
  Board = []

  print("current board is:\n")
  for r in range(len(Lines)):
    row = Lines[r].strip()
    print(row)

    if "G" in row:
      G = (r, row.replace(" ", "").find("G"))
    if "S" in row:
      if xs1==-1:
        xs1 = r
        ys1 = row.replace(" ", "").find("S")
        if row.count("S")==2:
          xs2 = r
          ys2 = row.replace(" ", "").find("S",ys1+1)

      else:
        xs2 = r
        ys2 = row.replace(" ", "").find("S")
```

```
    Board.append(row.split(" "))

  if xs2 == -1: #lies in z plane
    S = (xs1,ys1,xs1,ys1,"z")
  elif xs1==xs2: #lies in x plane
    S = (xs1,ys1,xs2,ys2,"x")
  else: #lies in y plane
    S = (xs1,ys1,xs2,ys2,"y")

  print("\nGoal is at:",G)
  print("Start is at:",S,"\n")

  return (S,G,Board)
```

1.2 1) Model the puzzle as a search problem: specify the states, successor state function, initial state, goal test, and step cost.

States, successor states and the goal state of the game depends on the position of our object and wheter it is vertical or horizontal.

We can define the state of object with 4 coordinate values easily and this would save us from specifying if the object is horizontal or vertical however i will still be indicating the object's status for ease of some calculations. We can think the object as 2 cubes merged and the 4 coordinates are x1, y1, x2 and y2 with x1,y1 and x2,y2 indicating the coordinates of two cubes over the board matrix ignoring the z-plane when object is vertical and showing it with another label to tell which of the x-y-z plane object lies across.

> For example the given object to us in the sample case of the homework document according to this state aproach is stated as **(2, 3, 3, 3, "y")** indicating the object is lying on the y-plane and its cubes are on the tiles (2,3) and (3,3). Similarly in the sample case the goal state is **(4, 7, 4, 7, "z")** indicating the object is vertiacl (lying on the z-plane) and its both cubes are on the tile (4,7).

Generally there are 4 successor states indicating the movements in 4 directions over x-y plane with up, down, left and right. However each move function has 3 different scenerios depending on which plane object lies along (Check the movement functions in the class definition for further information). And there are some movement restrictions such that object can't move if after movement at least one of its cubes will be out of the Board etc..

Initial state **(as an example is given above)** is just which tiles the object starts lying on which plane. And it can be easily extracted by reading the board matrix if there is a single S at a position x1,y1 then the initial state is (x1,y1,x1,y1) -object is at vertical at position (x1,y1)- and if there are two S at positions (x1,y1) and (x2,y2) the initial state is (x1,y1,x2,y2) -object is horizontal at position (x1,y1) and (x2,y2)-. Note that either x1 should be equal to x2 or y1 should be equal to y2 since the object cannot stand diagonally.

Assuming goal states are always 1 block wide -object need to stand on goal vertically-, goal test can be easily done by two steps. First define goal coordinate as a state **(as an example is given above)**, if goal is at position (xg,yg) then goal state is (xg,yg,xg,yg,"z"). Then as a second step we can easily check if the state we reached is goal state or not.

Finally as the game works with move count normally, we can accept each move as equal cost. Ultimately, converging UCS to BFS. **But a stepcost is set in the A* Search part according to the situation, please check there.**

Defining node as a class, including state, successor state generating functions and other functions:

[ ]:
```
class Node:
  def __init__ (self,x1,y1,x2,y2,status,previous=None):
    self.x1 = x1
    self.y1 = y1
    self.x2 = x2
    self.y2 = y2
    self.status = status # "z" = lies on z plane, "x" = lies on x plane
    #"y" = lies on y plane
    self.previous = previous

##### SHOW STATUS - FOR SELF CHECK
  def SHOW_ME_WHAT_YOU_GOT(self):
    print((self.x1,self.y1,self.x2,self.y2,self.status))

##### GET COORDINATES
  def get(self):
    return (self.x1,self.y1,self.x2,self.y2)

##### MOVE UP
  def move_up(self):
    Temp = Node(self.x1,self.y1,self.x2,self.y2,self.status,self)

    # when moving up from status "y", north point should move 1 and the other 2
    x1 = self.x1
    x2 = self.x2

    if self.status=="z":
      if (self.x1 - 2 >= 0) and (Board[self.x1-1][self.y1] != "X") and␣
 ↪(Board[self.x2-2][self.y2] != "X"):
        Temp.x1 -= 1
        Temp.x2 -= 2
        Temp.status = "y"
        return Temp

    elif self.status=="x":
      if (self.x1 - 1 >= 0) and (Board[self.x1-1][self.y1] != "X") and␣
 ↪(Board[self.x2-1][self.y2] != "X"):
        Temp.x1 -= 1
        Temp.x2 -= 1
        return Temp

    else: #self.status=="y"
```

```
      if (min(x1,x2) - 1 >= 0) and (Board[min(x1,x2) - 1][self.y1] != "X"):
        Temp.x1 = min(x1,x2)-1
        Temp.x2 = Temp.x1
        Temp.status = "z"
        return Temp

##### MOVE DOWN
  def move_down(self):
    Temp = Node(self.x1,self.y1,self.x2,self.y2,self.status,self)

    # when moving down from status "y", south point should move 1 and other 2
    x1 = self.x1
    x2 = self.x2

    if self.status=="z":
      if (self.x1 + 2 < len(Board)) and (Board[self.x1+1][self.y1] != "X") and␣
 ↪(Board[self.x2+2][self.y2] != "X"):
        Temp.x1 += 1
        Temp.x2 += 2
        Temp.status = "y"
        return Temp

    elif self.status=="x":
      if (self.x1 + 1 < len(Board)) and (Board[self.x1+1][self.y1] != "X") and␣
 ↪(Board[self.x2+1][self.y2] != "X"):
        Temp.x1 += 1
        Temp.x2 += 1
        return Temp

    else: #self.status=="y"
      if (max(x1,x2) + 1 < len(Board)) and (Board[max(x1,x2)+1][self.y1] !=␣
 ↪"X"):
        Temp.x1 = max(x1,x2)+1
        Temp.x2 = Temp.x1
        Temp.status = "z"
        return Temp

##### MOVE RIGHT
  def move_right(self):
    Temp = Node(self.x1,self.y1,self.x2,self.y2,self.status,self)

    # when moving right from status "x", east point should move 1 and other 2
    y1 = self.y1
    y2 = self.y2

    if self.status=="z":
```

```
      if (self.y1+2 < len(Board[0])) and (Board[self.x1][self.y1+1] != "X") and␣
↪(Board[self.x2][self.y2+2] != "X"):
        Temp.y1 += 1
        Temp.y2 += 2
        Temp.status = "x"
        return Temp

    elif self.status=="x":
      if (max(y1,y2) + 1 < len(Board[0])) and \
      (Board[self.x1][max(y1,y2)+1] != "X"):
        Temp.y1 = max(y1,y2)+1
        Temp.y2 = Temp.y1
        Temp.status = "z"
        return Temp

    else: #self.status=="y"
      if (self.y1 + 1 < len(Board[0])) and (Board[self.x1][self.y1+1] != "X")␣
↪and (Board[self.x2][self.y2+1] != "X"):
        Temp.y1 += 1
        Temp.y2 += 1
        return Temp

##### MOVE LEFT
  def move_left(self):
    Temp = Node(self.x1,self.y1,self.x2,self.y2,self.status,self)

    # when moving left from status "x", west point should move 1 and other 2
    y1 = self.y1
    y2 = self.y2

    if self.status=="z":
      if (self.y1 - 2 >= 0) and (Board[self.x1][self.y1-1] != "X") and␣
↪(Board[self.x2][self.y2-2] != "X"):
        Temp.y1 -= 1
        Temp.y2 -= 2
        Temp.status = "x"
        return Temp

    elif self.status=="x":
      if (min(y1,y2) - 1 >= 0) and (Board[self.x1][min(y1,y2)+1] != "X"):
        Temp.y1 = min(y1,y2)-1
        Temp.y2 = Temp.y1
        Temp.status = "z"
        return Temp

    else: #self.status=="y"
```

```
      if (self.y1 - 1 >= 0) and (Board[self.x1][self.y1-1] != "X") and␣
↪(Board[self.x2][self.y2-1] != "X"):
        Temp.y1 -= 1
        Temp.y2 -= 1
        return Temp

##### GOAL CHECK
  def is_goal(self, G):
    if (self.status == "z") and ((self.x1, self.y1) == G):
      return True
    return False
```

[ ]:
```
def success(node):

  Path = []
  if node.previous == None:
    print("Start and Goal is same")

  while node.previous != None:
    Path.insert(0,node)
    node = node.previous
  Path.insert(0,node)

  moves = len(Path)-1
  print("path found as following, with %s moves (including start at the␣
↪beginning and goal at the end):\n" %moves)

  for state in Path:
    print(state.get())
  print()
```

[ ]:
```
def isnot_visited(node,visited):
  for i in visited:
    # 2 cubes interpretation, as we don't track which cube is which we should
    # check their switched positions as they give the same state too
    # meaning positioning would be same for
    # two nodes (x1,y1,x2,y2) and (x2,y2,x1,y1) inreality

    # initially overridden equality symbol (__eq__) in class to check this
    # however it also overwrites not equal sign (__ne__)
    # and it interrupted my further check if node is not equal to None
    # I needed to implent it inside class and didn't want to spend time with it
    # and found writing 8 equality check here is easier

    if ((node.x1 == i.x1) and (node.x2 == i.x2) and (node.y1 == i.y1) and (node.
↪y2 == i.y2)) or \
    ((node.x1 == i.x2) and (node.x2 == i.x1) and (node.y1 == i.y2) and (node.y2␣
↪== i.y1)):
```

```
      return False
  return True
```

[ ]:
```
def insert_to_UCS(node,Q):
  if not (node in Q.queue):
    Q.put(node)
    return True # so that frontier size will be increased
  return False # so that frontier size Will NOT be increased
```

1.3 2) Implement in Python a UCS algorithm to solve the puzzle.

Note that, as the step cost is same for each of the move functions, the cost is not considered in the evaluation of the algorithm here converging to BFS algorithm

[ ]:
```
def UCS(S,G,Board):
  start_time = time.time()

  Start = Node(S[0],S[1],S[2],S[3],S[4])
  Q = queue.Queue()
  Q.put(Start)
  total_explored = 1

  visited = []

  while not Q.empty():
    A = Q.get()
    if A.is_goal(G):
      success(A)
      break

    visited.append(A)

    B = A.move_up()
    if (B!=None) and (isnot_visited(B,visited)):
    #if B == None executement will stop, so it is safe
      if insert_to_UCS(B,Q):
      #if node is insertable it will insert and return True
        total_explored += 1

    B = A.move_right()
    if (B!=None) and (isnot_visited(B,visited)):
      if insert_to_UCS(B,Q):
        total_explored += 1

    B = A.move_down()
    if (B!=None) and (isnot_visited(B,visited)):
      if insert_to_UCS(B,Q):
        total_explored += 1
```

```
    B = A.move_left()
    if (B!=None) and (isnot_visited(B,visited)):
      if insert_to_UCS(B,Q):
        total_explored += 1

  print("execution completed in %s seconds" % (time.time() - start_time))
  print("total frontier size reached (explored and saved nodes on memmory): %s"␣
↪% total_explored)
```

[ ]:
```
def UCS2(S,G,Board):
  start_time = time.time()

  Start = Node(S[0],S[1],S[2],S[3],S[4])
  Q = queue.Queue()
  Q.put(Start)
  total_explored = 1

  while not Q.empty():
    A = Q.get()
    if A.is_goal(G):
      success(A)
      break

    B = A.move_up()
    if (B!=None): #if B == None executement will stop, so it is safe
      Q.put(B)
      total_explored += 1

    B = A.move_right()
    if (B!=None):
      Q.put(B)
      total_explored += 1

    B = A.move_down()
    if (B!=None):
      Q.put(B)
      total_explored += 1

    B = A.move_left()
    if (B!=None):
      Q.put(B)
      total_explored += 1

  print("execution completed in %s seconds" % (time.time() - start_time))
  print("total frontier size reached (explored and saved nodes on memmory): %s"␣
↪% total_explored)
```

Search algorithm that also tracks already occured states, and avoids them:

[ ]:
```
A = ReadBoard("Board1.txt")
S = A[0]
G = A[1]
Board = A[2]
UCS(S,G,Board)
```

```
current board is:

O O O X X X X X X
O O O O O O X X X
O O O S O O O O X
X O O S O O O O O
X X X X X O O G O O
X X X X X X O O O X

Goal is at: (4, 7)
Start is at: (2, 3, 3, 3, 'y')

path found as following, with 5 moves (including start at the beginning and goal
at the end):

(2, 3, 3, 3)
(2, 4, 3, 4)
(2, 5, 3, 5)
(2, 6, 3, 6)
(2, 7, 3, 7)
(4, 7, 4, 7)

execution completed in 0.004397869110107422 seconds
total frontier size reached (explored and saved nodes on memmory): 78
```

Search algorithm that does NOT track the occurances:

[ ]:
```
A = ReadBoard("Board1.txt")
S = A[0]
G = A[1]
Board = A[2]
UCS2(S,G,Board)
```

```
current board is:

O O O X X X X X X
O O O O O O X X X
O O O S O O O O X
X O O S O O O O O
X X X X X O O G O O
X X X X X X O O O X
```

```
Goal is at: (4, 7)
Start is at: (2, 3, 3, 3, 'y')

path found as following, with 5 moves (including start at the beginning and goal
at the end):

(2, 3, 3, 3)
(2, 4, 3, 4)
(2, 5, 3, 5)
(2, 6, 3, 6)
(2, 7, 3, 7)
(4, 7, 4, 7)

execution completed in 0.007296085357666016 seconds
total frontier size reached (explored and saved nodes on memmory): 612
```

1.4 3) Extend your search model for A* search: Find a heuristic function and prove that it is admissible.

As we are on a discrete board with movement only in 4 base directions Euclidean Distance would be inconsistent.

Chebyshev Distance wouldn't be so reasonable since we have a totally discrete space.

For example let's assume goal is at position (x,y), (discarding rectangular property of object and thinking it as a cube) if we are away from the goal by 1 step in each direction A(x+1,y+1) or 1 step away in only one direction B(x+1,y); using Chebyshev Distance we will get same result for these 2 positions. However while state B requires only 1 move to reach the goal, state A requires 2 moves to reach the goal. Which is unreasonable.

Therefore, it would be better to use Manatthan Distance function with the following specification, when the object is horizontal and has 2 coordinates on the board, closest one to goal can be considered for distance calculation since the object moves from the closest point to given direction. Also note that this function will make sense in the following matter as well: The object only moves perpendicular to x and y axes, in otherwords it cannot move diagonally towards the goal, and therefore the object must go this total x ammount and y ammount seperately which matches with the idea of Manhattan Distance adding up the perpendicular distance from both axes.

Since we are setting step cost based on move count, setting this step cost 2 will be consistent; the object either moves 1 tile (2/3 of the all moves) or 2 tiles (1/3 of the all moves) whether it is vertical or horizontal.

It can be seen that going from one tile to other tile it takes **up to** total Manhattan Distance ammount of steps with either of the inital status (z,x,y);equal to diff(x) + diff(y) (which I will reffer as **distance** from now on).

> the maximum value of this real cost would be reached by each transition being 1 tile advancement towards the goal leading to **distance * step cost of 2**

the minimum value of this real cost would be reached by each transition being 2 tile advancement towards the goal leading to **distance * 0.5 * step cost of 2**.

Finally, since the heuristic function's estimated cost for a tile is always **distance** and the minimum **real cost** of that tile is **distance * 0.5 * step cost of 2** (note that in reality as if the obejct is not vertical the goal is not counted as reached therefore real value will be increased even more to adjust that with extra steps), which is equal to the heuristics, this holds the admissability requirement that **real cost will always be greater or equal to the heuristic value**, in other words heuristic value will not be an overestimate.

Note that it can be seen from here that for any step cost which is greater than or equal to 2 will suffice the condition where the real cost is greater than the estimation

Note that this also guarantees monotonity, namely increase in f values of the nodes as we get closer to the goal as following; 1. Distance between goal and start is x + y, for example if we are at state (x1, y1) then our remaining distance is (x - x1) + (y - y1) 2. we compute f values as (considering all transitions are 1 tile again), **g = (x1 + y1) * 2** and **h = (x - x1) + (y - y1)** 3. and therefore, **f = g + h = (x + x1) + (y + y1)** which indicates that as we move further towards the goal f values will increase accoring to increasing x1 and y1.

[ ]:
```
def h(node,G):
  a = abs(node.x1-G[0]) + abs(node.y1-G[1])
  b = abs(node.x2-G[0]) + abs(node.y2-G[1])
  return min(a,b)
```

[ ]:
```
def g(node):
  a = 0
  while node.previous != None:
    a += 1
    node = node.previous

  return a*2
```

[ ]:
```
def insert_to_ASS(newNodeTuple,Q):

  if Q.empty():
    Q.put(newNodeTuple)
    return

  tempNodeList = []
  while not Q.empty():
    tempNodeTuple = Q.get()

    #once again if nodes are at same position/status
    if ((newNodeTuple[2].x1 == tempNodeTuple[2].x1) and (newNodeTuple[2].x2 ==␣
↪tempNodeTuple[2].x2) and (newNodeTuple[2].y1 == tempNodeTuple[2].y1) and␣
↪(newNodeTuple[2].y2 == tempNodeTuple[2].y2)) or \
```

```
   ((newNodeTuple[2].x1 == tempNodeTuple[2].x2) and (newNodeTuple[2].x2 ==␣
↪tempNodeTuple[2].x1) and (newNodeTuple[2].y1 == tempNodeTuple[2].y2) and␣
↪(newNodeTuple[2].y2 == tempNodeTuple[2].y1)):

     if newNodeTuple[0] < tempNodeTuple[0]: #if same node found and if the␣
↪newly explored node has a lower cost
       Q.put(newNodeTuple) #put the new node instead of the higher cost one

     else:
     #if the same node in the queue has a lower or equal cost to the newly
     #explored node there is no need to put the new node to the queue as same
     #position will give same heuristic and if costs are same it means they
     #took equal moves as well. So, we only need to generate one valid␣
↪shortest
     #path from the begining to the goal, having multiple alternative paths
     #with same cost are unnecessary so I don't add the same cost

     #Put back the temporary element for next for loop below
       Q.put(tempNodeTuple)

     for tuples in tempNodeList:
     # revert the queue as we found the same node and took the required action
     # since this is a priority queue, we don't need to extract everything to
     # keep the sorting of the queue as elements are already inserted sorted
       Q.put(tuples)

     # return true so that total explored node count will be increased,
     # Either we change the old node with new one or discard the new one
     # we don't keep the track of the discarded one
     # and the saved node count in memmory is same
     return False

   else: #if the extracted element from queue is not same as the newly␣
↪explored node
   #just store the extracted node and continue iterating
     tempNodeList.append(tempNodeTuple)

 #if queue is emptied, no match found, put everything back along with new node
 #return true so that total explored node count will be increased

 for tuples in tempNodeList:
   Q.put(tuples)
 Q.put(newNodeTuple)
 return True
```

1.5 4) Implement in Python an A* search algorithm to solve the puzzle.

[ ]:
```
def ASS(S,G,Board):
  start_time = time.time()

  Start = Node(S[0],S[1],S[2],S[3],S[4])
  Q = queue.PriorityQueue()

  uniqueCounter = 0
  total_explored = 1

  Q.put((0,uniqueCounter,Start))
  #as this element will be popped as soon as we went in loop
  # putting it without cost does not affect the sorting
  # including uniqueCounter in the queue tuple due to the evaluating algorithm
  # of PriorityQueue() from queue library
  # it ensures that if there are 2 nodes with the same cost
  # firstly explored one will be expanded  first
  # (since priority queue will check the secon element of the tuple once
  #the first elements are same)
  visited = []


  while not Q.empty():
    A = Q.get()[2]
    if A.is_goal(G):
      success(A)
      break

    visited.append(A)

    B = A.move_up()
    if (B!=None) and (isnot_visited(B,visited)):
      f = g(B) + h(B,G)
      uniqueCounter += 1
      if insert_to_ASS((f,uniqueCounter,B),Q):
        total_explored += 1

    B = A.move_right()
    if (B!=None) and (isnot_visited(B,visited)):
      f = g(B) + h(B,G)
      uniqueCounter += 1
      if insert_to_ASS((f,uniqueCounter,B),Q):
        total_explored += 1

    B = A.move_down()
    if (B!=None) and (isnot_visited(B,visited)):
```

```
      f = g(B) + h(B,G)
      uniqueCounter += 1
      if insert_to_ASS((f,uniqueCounter,B),Q):
        total_explored += 1

    B = A.move_left()
    if (B!=None) and (isnot_visited(B,visited)):
      f = g(B) + h(B,G)
      uniqueCounter += 1
      if insert_to_ASS((f,uniqueCounter,B),Q):
        total_explored += 1

  print("execution completed in %s seconds" % (time.time() - start_time))
  print("total frontier size reached (explored and saved nodes on memmory): %s"␣
↪% total_explored)
```

[ ]:
```
def ASS2(S,G,Board):
  start_time = time.time()

  Start = Node(S[0],S[1],S[2],S[3],S[4])
  Q = queue.PriorityQueue()

  total_explored = 1
  Q.put((0,total_explored,Start))

  while not Q.empty():
    A = Q.get()[2]
    if A.is_goal(G):
      success(A)
      break

    B = A.move_up()
    if (B!=None):
      f = g(B) + h(B,G)
      total_explored += 1
      Q.put((f,total_explored,B))

    B = A.move_right()
    if (B!=None):
      f = g(B) + h(B,G)
      total_explored += 1
      Q.put((f,total_explored,B))

    B = A.move_down()
    if (B!=None):
      f = g(B) + h(B,G)
      total_explored += 1
```

```
      Q.put((f,total_explored,B))

    B = A.move_left()
    if (B!=None):
      f = g(B) + h(B,G)
      total_explored += 1
      Q.put((f,total_explored,B))

  print("execution completed in %s seconds" % (time.time() - start_time))
  print("total frontier size reached (explored and saved nodes on memmory): %s"␣
↪% total_explored)
```

Search algorithm that also tracks already occured states, and avoids them:

[ ]:
```
A = ReadBoard("Board1.txt")
S = A[0]
G = A[1]
Board = A[2]
ASS(S,G,Board)
```

```
current board is:

O O O X X X X X X
O O O O O O X X X
O O O S O O O O X
X O O S O O O O O
X X X X X O O G O
X X X X X X O O X

Goal is at: (4, 7)
Start is at: (2, 3, 3, 3, 'y')

path found as following, with 5 moves (including start at the beginning and goal
at the end):

(2, 3, 3, 3)
(2, 4, 3, 4)
(2, 5, 3, 5)
(2, 6, 3, 6)
(2, 7, 3, 7)
(4, 7, 4, 7)

execution completed in 0.002227306365966797 seconds
total frontier size reached (explored and saved nodes on memmory): 23
```

Search algorithm that does NOT track the occurances:

[ ]:
```
A = ReadBoard("Board1.txt")
S = A[0]
G = A[1]
```

```
Board = A[2]
ASS2(S,G,Board)
```

```
current board is:

O O O X X X X X X
O O O O O O X X X
O O O S O O O O O X
X O O S O O O O O O
X X X X X O O G O O
X X X X X X O O O X

Goal is at: (4, 7)
Start is at: (2, 3, 3, 3, 'y')

path found as following, with 5 moves (including start at the beginning and goal
at the end):

(2, 3, 3, 3)
(2, 4, 3, 4)
(2, 5, 3, 5)
(2, 6, 3, 6)
(2, 7, 3, 7)
(4, 7, 4, 7)

execution completed in 0.002330303192138672 seconds
total frontier size reached (explored and saved nodes on memmory): 53
```

1.6 5) Compare your UCS and A* codes on some sample puzzle instances. Construct a table that shows, for each instance, time and memory consumption. Discuss the results of these experiments: Are the results surprising or as expected? Please explain.

1.6.1 Other Test Cases

Board2

[ ]:
```
A = ReadBoard("Board2.txt")
S = A[0]
G = A[1]
Board = A[2]
```

```
current board is:

O O O X X X X X X
S S O O O O X X X X
O O O O O O O O O X
X O O O O O O G O O
```

```
X X X X X O O O O O
X X X X X X O O O X

Goal is at: (3, 7)
Start is at: (1, 0, 1, 1, 'x')
```

[ ]:
```
print("UCS algorithm with visited tile tracking")
UCS(S,G,Board)
```

```
UCS algorithm with visited tile tracking
path found as following, with 8 moves (including start at the beginning and goal
at the end):

(1, 0, 1, 1)
(0, 0, 0, 1)
(0, 2, 0, 2)
(1, 2, 2, 2)
(1, 3, 2, 3)
(1, 4, 2, 4)
(3, 4, 3, 4)
(3, 5, 3, 6)
(3, 7, 3, 7)

execution completed in 0.013544797897338867 seconds
total frontier size reached (explored and saved nodes on memmory): 152
```

[ ]:
```
print("UCS algorithm with NO visited tile tracking")
UCS2(S,G,Board)
```

```
UCS algorithm with NO visited tile tracking
path found as following, with 8 moves (including start at the beginning and goal
at the end):

(1, 0, 1, 1)
(0, 0, 0, 1)
(0, 2, 0, 2)
(1, 2, 2, 2)
(1, 3, 2, 3)
(1, 4, 2, 4)
(3, 4, 3, 4)
(3, 5, 3, 6)
(3, 7, 3, 7)

execution completed in 0.03739333152770996 seconds
total frontier size reached (explored and saved nodes on memmory): 5272
```

[ ]:
```
print("A* Search algorithm with visited tile tracking")
ASS(S,G,Board)
```

```
A* Search algorithm with visited tile tracking
path found as following, with 8 moves (including start at the beginning and goal
at the end):

(1, 0, 1, 1)
(0, 0, 0, 1)
(0, 2, 0, 2)
(1, 2, 2, 2)
(1, 3, 2, 3)
(1, 4, 2, 4)
(3, 4, 3, 4)
(3, 5, 3, 6)
(3, 7, 3, 7)

execution completed in 0.01138615608215332 seconds
total frontier size reached (explored and saved nodes on memmory): 80
```

[ ]:
```
print("A* Search algorithm with NO visited tile tracking")
ASS2(S,G,Board)
```

```
A* Search algorithm with NO visited tile tracking
path found as following, with 8 moves (including start at the beginning and goal
at the end):

(1, 0, 1, 1)
(0, 0, 0, 1)
(0, 2, 0, 2)
(1, 2, 2, 2)
(1, 3, 2, 3)
(1, 4, 2, 4)
(3, 4, 3, 4)
(3, 5, 3, 6)
(3, 7, 3, 7)

execution completed in 0.01732802391052246 seconds
total frontier size reached (explored and saved nodes on memmory): 1296
```

Board3

[ ]:
```
A = ReadBoard("Board3.txt")
S = A[0]
G = A[1]
Board = A[2]
```

```
current board is:

O O G X X X X X X X
O O O O O O X X X X
O O O O O O O O O X
X O O O O O O O O O
X X X X X O O O S O
X X X X X X O O S X

Goal is at: (0, 2)
Start is at: (4, 8, 5, 8, 'y')
```

[ ]:
```
print("UCS algorithm with visited tile tracking")
UCS(S,G,Board)
```

```
UCS algorithm with visited tile tracking
path found as following, with 7 moves (including start at the beginning and goal
at the end):

(4, 8, 5, 8)
(3, 8, 3, 8)
(3, 7, 3, 6)
(3, 5, 3, 5)
(3, 4, 3, 3)
(3, 2, 3, 2)
(2, 2, 1, 2)
(0, 2, 0, 2)

execution completed in 0.00784611701965332 seconds
total frontier size reached (explored and saved nodes on memmory): 123
```

[ ]:
```
print("UCS algorithm with NO visited tile tracking")
UCS2(S,G,Board)
```

```
UCS algorithm with NO visited tile tracking
path found as following, with 7 moves (including start at the beginning and goal
at the end):

(4, 8, 5, 8)
(3, 8, 3, 8)
(3, 7, 3, 6)
(3, 5, 3, 5)
(3, 4, 3, 3)
(3, 2, 3, 2)
(2, 2, 1, 2)
(0, 2, 0, 2)
```

```
execution completed in 0.021177053451538086 seconds
total frontier size reached (explored and saved nodes on memmory): 2475
```

[ ]:
```
print("A* Search algorithm with visited tile tracking")
ASS(S,G,Board)
```

```
A* Search algorithm with visited tile tracking
path found as following, with 7 moves (including start at the beginning and goal
at the end):

(4, 8, 5, 8)
(3, 8, 3, 8)
(3, 7, 3, 6)
(3, 5, 3, 5)
(3, 4, 3, 3)
(3, 2, 3, 2)
(2, 2, 1, 2)
(0, 2, 0, 2)

execution completed in 0.009929418563842773 seconds
total frontier size reached (explored and saved nodes on memmory): 50
```

[ ]:
```
print("A* Search algorithm with NO visited tile tracking")
ASS2(S,G,Board)
```

```
A* Search algorithm with NO visited tile tracking
path found as following, with 7 moves (including start at the beginning and goal
at the end):

(4, 8, 5, 8)
(3, 8, 3, 8)
(3, 7, 3, 6)
(3, 5, 3, 5)
(3, 4, 3, 3)
(3, 2, 3, 2)
(2, 2, 1, 2)
(0, 2, 0, 2)

execution completed in 0.004758596420288086 seconds
total frontier size reached (explored and saved nodes on memmory): 109
```

Board4

[ ]:
```
A = ReadBoard("Board4.txt")
S = A[0]
G = A[1]
Board = A[2]
```

```
current board is:

O O O X X X X X X
O O O O O O X X X X
O O G O O O O O O X
X O O O O O O O O O
X X X X X O O O O O
X X X X X X O O O X
X S X X X O O O O X
X O X O O O O O O O
X O O O O O O O O O
X O O O O O O O O O

Goal is at: (2, 2)
Start is at: (6, 1, 6, 1, 'z')
```

[ ]:
```
print("UCS algorithm with visited tile tracking")
UCS(S,G,Board)
```

```
UCS algorithm with visited tile tracking
path found as following, with 14 moves (including start at the beginning and
goal at the end):

(6, 1, 6, 1)
(7, 1, 8, 1)
(9, 1, 9, 1)
(9, 2, 9, 3)
(9, 4, 9, 4)
(8, 4, 7, 4)
(8, 5, 7, 5)
(6, 5, 6, 5)
(6, 6, 6, 7)
(5, 6, 5, 7)
(5, 5, 5, 5)
(4, 5, 3, 5)
(2, 5, 2, 5)
(2, 4, 2, 3)
(2, 2, 2, 2)

execution completed in 0.03546595573425293 seconds
total frontier size reached (explored and saved nodes on memmory): 397
```

[ ]:
```
print("UCS algorithm with NO visited tile tracking")
UCS2(S,G,Board)
```

```
UCS algorithm with NO visited tile tracking
path found as following, with 14 moves (including start at the beginning and
```

```
goal at the end):

(6, 1, 6, 1)
(7, 1, 8, 1)
(9, 1, 9, 1)
(9, 2, 9, 3)
(9, 4, 9, 4)
(8, 4, 7, 4)
(8, 5, 7, 5)
(6, 5, 6, 5)
(6, 6, 6, 7)
(5, 6, 5, 7)
(5, 5, 5, 5)
(4, 5, 3, 5)
(2, 5, 2, 5)
(2, 4, 2, 3)
(2, 2, 2, 2)

execution completed in 2.7503998279571533 seconds
total frontier size reached (explored and saved nodes on memmory): 391914
```

[ ]:
```
print("A* Search algorithm with visited tile tracking")
ASS(S,G,Board)
```

```
A* Search algorithm with visited tile tracking
path found as following, with 14 moves (including start at the beginning and
goal at the end):

(6, 1, 6, 1)
(7, 1, 8, 1)
(9, 1, 9, 1)
(9, 2, 9, 3)
(9, 4, 9, 4)
(8, 4, 7, 4)
(8, 5, 7, 5)
(6, 5, 6, 5)
(6, 6, 6, 7)
(5, 6, 5, 7)
(5, 5, 5, 5)
(4, 5, 3, 5)
(2, 5, 2, 5)
(2, 4, 2, 3)
(2, 2, 2, 2)

execution completed in 0.03283953666687012 seconds
total frontier size reached (explored and saved nodes on memmory): 145
```

```
[ ]: print("A* Search algorithm with NO visited tile tracking")
     ASS2(S,G,Board)
```

```
A* Search algorithm with NO visited tile tracking
path found as following, with 14 moves (including start at the beginning and
goal at the end):

(6, 1, 6, 1)
(7, 1, 8, 1)
(9, 1, 9, 1)
(9, 2, 9, 3)
(9, 4, 9, 4)
(8, 4, 7, 4)
(8, 5, 7, 5)
(6, 5, 6, 5)
(6, 6, 6, 7)
(5, 6, 5, 7)
(5, 5, 5, 5)
(4, 5, 3, 5)
(2, 5, 2, 5)
(2, 4, 2, 3)
(2, 2, 2, 2)

execution completed in 0.049627065658569336 seconds
total frontier size reached (explored and saved nodes on memmory): 5063
```

1.6.2 Results for all test cases

Case	Algorithm	Runing time in seconds	#of total explored nodes	#of moves taken to reach the goal
Board1	UCS with tile tracking	~0.0036	78	5
Board1	UCS with no tile tracking	~0.0098	612	5
Board1	A* with tile tracking	~0.0031	23	5

Case	Algorithms	Runing time in seconds	#of total explored nodes	#of moves taken to reach the goal
Board1	A* with no tile tracking	~0.0024	53	5

Case	Algorithms	Runing time in seconds	#of total explored nodes	#of moves taken to reach the goal
Board2	UCS with tile tracking	~0.0088	152	8
Board2	UCS with no tile tracking	~0.0812	5272	8
Board2	A* with tile tracking	~0.0126	80	8
Board2	A* with no tile tracking	~0.0192	1296	8

Case	Algorithms	Runing time in seconds	#of total explored nodes	#of moves taken to reach the goal
Board3	UCS with tile tracking	~0.0064	123	7
Board3	UCS with no tile tracking	~0.0231	2475	7
Board3	A* with tile tracking	~0.0082	50	7
Board3	A* with no tile tracking	~0.0023	109	7

Case	Algorithms	Runing time in seconds	#of total explored nodes	#of moves taken to reach the goal
Board4	UCS with tile tracking	~0.0477	397	14
Board4	UCS with no tile tracking	~2.6875	391914	14
Board4	A* with tile tracking	~0.0322	145	14

Case	Algorithms	Runing time in seconds	#of total explored nodes	#of moves taken to reach the goal
Board4	A* with no tile tracking	~0.0485	5063	14

Some remarks

Firstly want to indicate that since the runing time of the algorithms, as it should be expected, changes each time they are run by a very little ammount (around 3ms); while this ammount is insignificantly small, since the runtimes of the algorithms are also very short (even shorter in smaller cases) this sometimes messes with the comparison of times however there are some significant results.

Second, A* algorithm with tile tracking has more condition checks than all other algorithms here. Therefore it is time-wise left behind other algorithms when statespace is smaller, where brute force of no tile tracking beats the smartness of tile tracking with no time wasted in condition checks.

And lastly all algorithms are evaluated with the UP, RIGHT, DOWN, LEFT expansion order.

Conclusion

AS it can be expected, A* Search has always given better results when comparing the space consumed to store the explored nodes compared to UCS. Also notice that, as the depth increased (move count, depth of goal in the search space), the gap between the performance of A* search and UCS algorithms became unignorably huge.

In the tracking versions of algorithms, as the number of nodes to be explored reduced runing time of the A* search was no faster than UCS algorithm (as explained above, also see in the table for Board1, 2 and 3); however as the space size increased (there wasn't a significant increase in our test cases but it can be still seen in Board4), the runing time of the A* search started improving against UCS.

In non-tracking versions, the results are even more devastating for the sake of UCS algorithm. A* Search dramatically reduces the space used for the problem; infact as the depth increases, the space used by non-tracking A* search decreases exponentially compared to the non-tracking UCS. Additionally as condition checks are not included, A* search is not slowed down, the running time of the A* search was better than UCS even in the smaller depths/search spaces.

Without a doubt, tracking version of the both algorithms can always be considered better than their non-tracking counterparts both by an expection before the testing and by the results of the testing. Finally, tracking A* search algorithm can be accepted as the best out of these four algorithms for this problem. Even if it was slower in than some other algorithms in some of the cases, the time difference was always negligible; Additionally it always gave the least space occupation across all of the four algorithms in all cases.